FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge

Cong Hao[1*], Xiaofan Zhang[1*], Yuhong Li[1], Sitao Huang[1], Jinjun Xiong[2], Kyle Rupnow[3], Wen-mei Hwu[1], Deming Chen[1,3]
[1]University of Illinois at Urbana-Champaign, [2]IBM T. J. Watson Research Center, [3]Inspirit IoT, Inc.
{congh, xiaofan3, leeyh, shuang91, w-hwu, dchen}@illinois.edu, jinjun@us.ibm.com, kyle.rupnow@inspirit-iot.com

ABSTRACT

While embedded FPGAs are attractive platforms for DNN acceleration on edge-devices due to their low latency and high energy efficiency, the scarcity of resources of edge-scale FPGA devices also makes it challenging for DNN deployment. In this paper, we propose a simultaneous FPGA/DNN co-design methodology with both bottom-up and top-down approaches: a bottom-up hardware-oriented DNN model search for high accuracy, and a top-down FPGA accelerator design considering DNN-specific characteristics. We also build an automatic co-design flow, including an *Auto-DNN* engine to perform hardware-oriented DNN model search, as well as an *Auto-HLS* engine to generate synthesizable C code of the FPGA accelerator for explored DNNs. We demonstrate our co-design approach on an object detection task using PYNQ-Z1 FPGA. Results show that our proposed DNN model and accelerator outperform the state-of-the-art FPGA designs in all aspects including Intersection-over-Union (IoU) (6.2% higher), frames per second (FPS) (2.48× higher), power consumption (40% lower), and energy efficiency (2.5× higher). Compared to GPU-based solutions, our designs deliver similar accuracy but consume far less energy.

1 INTRODUCTION

The world has seen rapid adoption of FPGAs for DNN acceleration [1–6]. Internet of Things (IoT) applications in domains such as self-driving, security and surveillance face particular challenges as they require both sophisticated DNN models for Quality of Results (QoR) and strict latency, power, and resource constraints. Embedded FPGAs are one of the most attractive candidates to enable machine learning capability for IoT applications [7] because of their high energy efficiency and low cost, but the scarcity of resources also makes DNN accelerator design and deployment on FPGA more challenging. In a typical top-down design flow, DNN models are first designed concentrating more on the QoR, expecting the accelerator can meet performance constraints through later optimization. This approach has been largely successful, but ignores the impact that deployment architecture should have on the DNN design. Instead, DNNs should be built bottom-up with adequate understanding of the hardware constraints before expanding network size to reach the targeted QoR. Most importantly, DNNs and the corresponding FPGA accelerators need to be developed simultaneously, and we believe in FPGA/DNN co-design as a promising solution with immense optimization opportunity: DNN designs should be FPGA-architecture driven, and FPGA accelerators should be DNN-aware.

Despite the opportunities, a good co-design approach requires the exploration of an extremely large number of variables in the combined DNN and FPGA accelerator co-design space, and constrains the solutions to have both high QoR and efficient FPGA implementations. Consequently, the co-design task will be extremely time-consuming, as we must perform training of each candidate DNN to evaluate its quality. Even using Neural Architecture Search (NAS) [8, 9] for DNN development and the High Level Synthesis (HLS) for fast FPGA development [10, 11], both tasks still need a large amount of engineering hours.

Facing the opportunities and challenges, in this work, we propose a simultaneous FPGA/DNN co-design approach, which effectively searches the design space to both generate high quality DNNs suitable for FPGA deployment, and highly optimized FPGA accelerators. The contributions are summarized as follows:

- We propose the first simultaneous FPGA/DNN co-design methodology with (1) hardware-oriented DNN model design following bottom-up approach, and (2) DNN-driven FPGA accelerator design following top-down approach. A fully automatic co-design flow is developed accordingly for simultaneous DNN search and FPGA accelerator generation.
- For DNN model design, we introduce a DNN template to guide the DNN generation with predictable performance and resource utilization, which greatly reduces the co-design search space. Based on such template, an automatic DNN model search engine, *Auto-DNN*, is proposed to effectively explore the design space and generate DNN models for desired QoR.
- For FPGA accelerator design, we introduce a fine-grained tile-based pipeline architecture, which supports arbitrary DNNs generated by *Auto-DNN* using a library of highly optimized HLS IPs. Based on such architecture, an automatic HLS generator, *Auto-HLS*, is proposed to directly generate synthesizable C code of the DNN models, to conduct latency/resource estimation and FPGA accelerator generation.
- We demonstrate our co-design approach on an object detection task targeting a PYNQ-Z1 embedded FPGA. DNN models are searched and mapped to the board with the state-of-the-art performance regarding accuracy, speed, and power efficiency.

2 RELATED WORK

DNN model design and FPGA accelerator design are each under intense study, but these activities are often conducted independently. DNN design is conducted either manually by machine learning experts or automatically by Neural Architecture Search (NAS) such as recursive neural networks (RNN) [8] and reinforcement learning [12]. Although high QoR can be obtained, the DNNs may have complex structures that are unsuitable for FPGA deployment. A few platform-aware DNN search methods are proposed, such as [13, 14], but they only consider the DNN inference latency on CPUs and GPUs, not on FPGAs. On the other hand, for FPGA-based DNN accelerator, recent technologies such as quantization [2, 4] and model compression [15] are used to reduce DNN model size, and latency-directed resource allocation [3] and fine-grained pipeline architecture [5] are proposed to deliver low latency during DNN

*These authors made equal contributions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317829

Table 1: Key Variables for FPGA/DNN Co-Design

Variables	Explanation	Effect
L	Total number of layers	A, P, R
$IP_1, IP_2, \cdots, IP_m$	IP templates for DNN building	A, P, R
$p_1, p_2, \cdots, p_n$	Labels for IP instances	P, R
$\langle PF_j, Q_j \rangle$	Configuration for $p_j (1 \leq j \leq n)$	A, P, R
$\langle l_j^1, \cdots, l_j^z \rangle$	The layers where p_j is used	A, P
$< f_{ch_1}, f_{ch_2}, \cdots, f_{ch_L} >$	Channel expansion factors	A, P, R
$ds_1, ds_2, \cdots, ds_k$	Down-sampling layers	A, P, R
f_{ds_j}	Down-sampling factor	A, P, R
A: Accuracy, P: Performance, R: Resource		

inference. However, these approaches may be limited by the DNN models, and may not have sufficient optimization opportunities to meet performance constraints on target IoT platforms. Other works specifically conduct design space exploration to select hardware configuration parameters [16, 17] together with optimizations including loop unrolling and pipelining, but they do not explore configurations on the DNN side, which could make hardware implementations more effective.

3 FPGA/DNN CO-DESIGN

3.1 Co-Design Space

There is a large design space for DNN design, such as the number and types of layers, the number of input/output channels, residual connections, concatenations, etc. Similarly, the design space for FPGA accelerator is also enormous, such as IP instance categories, IP reuse strategies, quantization schemes, parallel factors, data transfer behaviors, and buffer sizes, etc. Thus, to cover both DNN model and accelerator design, the co-design space is exponentially greater than any of the above, which requires effective techniques to find high quality solutions. In this work, we conduct FPGA/DNN exploration by proposing a *co-design space* to efficiently narrow down the effort for space searching.

The variables in the proposed co-design space are summarized in Table 1. For FPGA accelerator, we use IP-based design strategy as in [3, 5]. Each IP supports a basic DNN layer type (e.g. Conv, Pooling), which must be instantiated and configured if the DNN model contains such type of layer. L is the total number of DNN layers. IP_1 to IP_m represent the available configurable IP templates. $p_j (1 \leq j \leq n)$ represent the configured IP instances, where the configurable parameters include parallelism factor PF_j and quantization scheme Q_j. $< l_j^1, \cdots l_j^z >$ represents the layers for which an IP instance p_j is used in FPGA to conduct the computation. Vector $< f_{ch_1}, f_{ch_2}, \cdots, f_{ch_L} >$ represents the expansions of channel depth through the entire DNN. In addition, ds_1 to ds_k represent down sampling layers with a down sampling factor f_{ds_i}. The combination of these parameters can specify both the DNN model and the accelerator design.

3.2 Overall Co-Design Flow

Our co-design flow solves two design problems simultaneously: the bottom-up DNN model exploration, and the top-down FPGA accelerator generation. For DNN models, we start from basic hardware-aware building blocks, and gradually construct DNNs to reach desired QoR; for FPGA accelerators, we follow a fixed architecture, and optimize configurable parameters to pursue most efficient DNN implementations. Regarding the two tasks, we propose the following four key components:

- For DNN: (1) **Bundle-Arch**: a hardware-aware DNN building block template to build up DNN models; (2) **Auto-DNN**: an efficient search engine to explore DNN candidates under hardware resource and performance constraints;
- For FPGA: (3) **Tile-Arch**: a low-latency FPGA accelerator template for DNN implementation; (4) **Auto-HLS**: a fast board-level design generator to automatically map DNNs onto FPGAs.

These four components work seamlessly as: *Auto-DNN* generates DNNs using *Bundle-Arch* DNN templates, while *Auto-HLS* builds accelerators following the *Tile-Arch* FPGA template. Meanwhile, *Auto-DNN* and *Auto-HLS* execute iteratively for DNN model search and FPGA accelerator generation.

Fig. 1 shows the overall flow of our proposed co-design methodology, composed of the four key components. The inputs include: targeted machine learning task (e.g., classification, detection), target FPGA device with resource constraints (e.g., DSP, LUTs, memory), and the performance targets of the accelerator (e.g., latency). We also have configurable IP templates as inputs. The outputs include hardware-oriented DNN models and their FPGA accelerators. There are three major steps in our co-design flow:

(1) **Co-Design Step 1: Building block and DNN modeling**. Given DNN building blocks and hardware IP pool, we first construct analytical models to capture the hardware latency and resource utilization of the building blocks and the DNNs built from the blocks. This is to provide performance estimation in the early stage of DNN exploration.
(2) **Co-Design Step 2: Building block selection**. To select the most promising DNN building blocks for the specific machine learning task and target FPGA, *Auto-DNN* performs both coarse- and fine-grained evaluations of the building blocks regarding three most important features: latency, resource utilization and accuracy. Based on the evaluation, building blocks on the Pareto curve will be selected for further DNN exploration.
(3) **Co-Design Step 3: Hardware-aware DNN search and update**. Given selected building blocks, *Auto-DNN* explores the DNNs under given resource and latency constraints by using stochastic coordinate descent (SCD). DNNs output from SCD are passed to *Auto-HLS* to get more precise performance and resource results, and are fed back to SCD for update. The generated DNNs that meet performance and resource requirements are output for training and fine-tuning.

In the following, the *Bundle-Arch* and *Tile-Arch* templates are introduced in Sec. 4. Building block evaluation and DNN search using *Auto-DNN* and *Auto-HLS* are introduced in Sec. 5.

4 DNN AND ACCELERATOR TEMPLATE

4.1 Bundle-Arch: Hardware-Aware DNN Template

We use DNN templates for model exploration because: (1) they help narrow down the DNN design space and speedup the search process, and (2) they can integrate hardware knowledge and guide DNN design towards hardware-oriented directions.

We propose a hardware-aware *Bundle* based DNN template, *Bundle-Arch*. A Bundle is a set of sequential DNN layers as a basic DNN building block. For example, a Bundle *i*-r_1 in Fig. 2 contains four DNN layers cascaded from top to bottom. DNN models are built by replicating, shaping and configuring a Bundle in a bottom-up

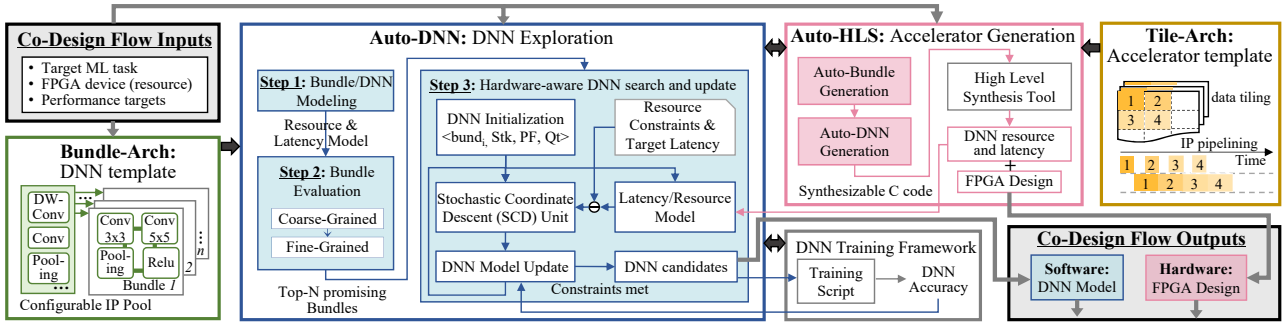


Figure 1: The overall FPGA/DNN co-design flow is composed of four key components: *Bundle-Arch* as a hardware-aware DNN template (green); *Auto-DNN* for DNN exploration (blue); *Auto-HLS* for FPGA accelerator synthesizable C code generation (pink); *Tile-Arch* as a low-latency accelerator template (yellow). Auto-DNN works as the primary component and outputs DNN models, while Auto-HLS outputs the corresponding FPGA implementations of the DNN models.

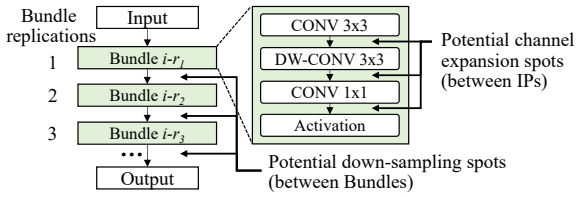


Figure 2: Bundle-Arch: A DNN template using pre-designed hardware-aware Bundles.

manner. In Fig. 2, three replications of the same Bundle are shown, where each replication may vary in input/output data dimensions. Between Bundles, we reserve down-sampling spots for feature map size compression. When implemented on FPGA, a hardware Bundle also represents a combination of the IP instances used for DNN layer computation. The IPs within one Bundle are organized based on our proposed *Tile-Arch* (in Sec.4.3), which delivers optimized low-latency designs.

We adopt the Bundle based strategy for building hardware-oriented DNNs following the same trend of modern popular DNNs, such as the residual block in ResNet [18] and depth-wise blocks in Mobilenet [19]. Moreover, FPGA accelerators can also benefit from pre-designed and optimized hardware Bundles, which provide more predictable patterns on computation and memory access behaviors.

4.2 Bundle Generation

To generate DNN Bundles, we select the following IPs (DNN layers) similar to previous NAS works as: convolution (conv) 1×1, 3×3, 5×5; depth-wise conv 3×3, 5×5, 7×7; max/avg pooling; normalization; activation. In FPGA implementation, each IP requires at least one instance, and more IPs mean more resource overhead. In this work, we limit up to two computational IPs in each Bundle since we are targeting IoT devices with scarce resources. It can be easily extended to support more IPs for devices with more resources.

In our experiments, 18 Bundle candidates are generated offline and used for DNN exploration. However, as we have more IPs, the number of Bundles may grow significantly. For scalability, the Bundles will be evaluated first (in Sec. 5.1), and the most promising ones will be selected for further DNN exploration based on their potential accuracy contributions and hardware characteristics.

4.3 Tile-Arch: Low Latency Accelerator Template

We propose a fine-grained tile-based pipeline accelerator architecture template, *Tile-Arch*, for mapping DNNs onto embedded FPGAs, which can deliver low latency designs and exploit maximum resource saving. This template has the following features:

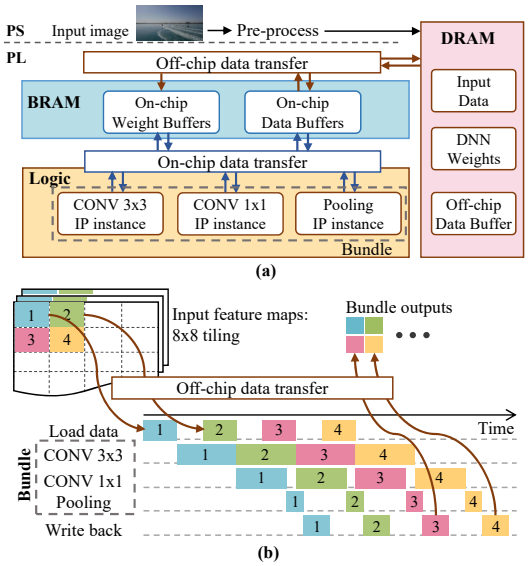


Figure 3: Tile-Arc: a low latency FPGA accelerator template with (a) a top-level diagram of the proposed architecture and (b) an example of tile-based pipeline structure.

- Layer-level IP reuse: we adopt a folded overall structure, where the DNN layers are computed sequentially on FPGA by reusing IP instances across layers. It can maximally exploit resource reuse, which is especially crucial for embedded FPGAs.
- Tile-level IP reuse: resulting from layer-level IP reuse, the intermediate data between layers are partitioned into tiles of common size across all layers, and an IP instance is reused for multiple tiles. It allows direct data transfer between IP instances of subsequent layers without on-/off-chip memory access.
- Tile-level pipelining: since data tiles within a layer do not have data dependencies, we can leverage tile-level IP pipelining both within a layer and across consecutive layers.

Fig. 3 (a) shows an example of the top-level diagram of the proposed template architecture. In this example, the Bundle contains IP instances including conv 3×3, 1×1 and pooling. On-chip data buffers are allocated in BRAM for intra-Bundle communication, while off-chip data buffers are allocated in DRAM for inter-Bundle communication. Fig. 3 (b) illustrates the tile-level pipelining for computation in one Bundle with four tiles. Following the top-down approach, parameters of the proposed architecture can be configured to adapt to different FPGA devices and to maximize the performance of FPGA accelerators.

4.4 Bundle and DNN Performance Modeling

Based on the proposed *Tile-Arch*, we build analytical models for performance and resource estimation for both Bundles and DNNs used in Bundle evaluation and DNN exploration. In this work, we take latency as the primary performance measure.

4.4.1 Bundle Performance Modeling. Denoted a Bundle as $bund_i$, the resource of $bund_i$ is computed as:

$$Res^r_{bund_i} = \sum_{p_j} Res^r_j + \Gamma^r_i \quad (1)$$

where Res^r_j is the resource usage of instance p_j of resource type r (including DSP, LUTs, FF and BRAM). Γ^r_i represents other resource overhead such as LUTs consumed by control logic and multiplexers.

The latency of a Bundle is estimated as:

$$Lat_{bund_i} = \alpha_i \cdot \sum_{p_j} Comp_j + \frac{\beta_i \cdot \Theta(Data_i)}{bw} \quad (2)$$

where $Comp_j$ is the computation latency of instance p_j, and $\Theta(Data_i)$ is the data amount processed by $bund_i$. bw represents the off-chip memory bandwidth. Denote the latency of one execution of p_j as lat_j, and the total number of reuses of p_j as $reuse_j$, the computation latency $Comp_j$ is estimated as:

$$Comp_j = \sum_{1 \le j \le n} reuse_j \cdot lat_j \quad (3)$$

$reuse_j$ can be computed by the input/output dimensions of the data processed by the IP and the data dimensions of p_j's interface. The parameter α_i in Eq. 2 describes how much computation is overlapped because of IP pipelining, and β_i describes how much data transfer is overlapped during computations. α_i, β_i and Γ_i will be determined for each $bund_i$ using *Auto-HLS* sampling.

4.4.2 DNN Performance Modeling. The overall DNN latency based on Lat_{bund_i} in Eq. 2 is estimated as:

$$Lat_{DNN} = \sum_{i=1}^{N} Lat_{bund} + \phi \cdot Lat_{DM} \quad (4)$$

where N is the the number of Bundle repetitions of the DNN, and $\phi \cdot Lat_{DM}$ represents the inter-bundle data movement latency. For overall DNN resource utilization, we have:

$$Res_{DNN} = Res_{bund_i} + \gamma \cdot Res_{ctl} \quad (5)$$

where Res_{bund_i} is the resource of $bund_i$, and Res_{ctl} is additional control logic overhead, e.g., finite state machine and multiplexers. ϕ, γ, Lat_{DM} and Res_{ctl} will be decided through *Auto-HLS* sampling.

5 DNN EXPLORATION AND UPDATE

The DNN exploration and update is conducted by *Auto-DNN* co-operated with *Auto-HLS*. Given a specific machine learning task, coarse- and fine-grained Bundle evaluation is first performed to select the top-N promising candidates. After that, a hardware-aware DNN exploration and update is performed, to search for DNNs within hardware resource and latency constraints. To better illustrate our approach, we use an object detection task specified by the 2018 Design Automation Conference System Design Contest (DAC-SDC) [20] as an example. This competition targets implementing machine learning applications on an embedded PYNQ-Z1 FPGA (with 4.9Mbit on-chip BRAM, 220 DSPs, 53,200 LUTs and 106,400 FFs) for board-level designs.

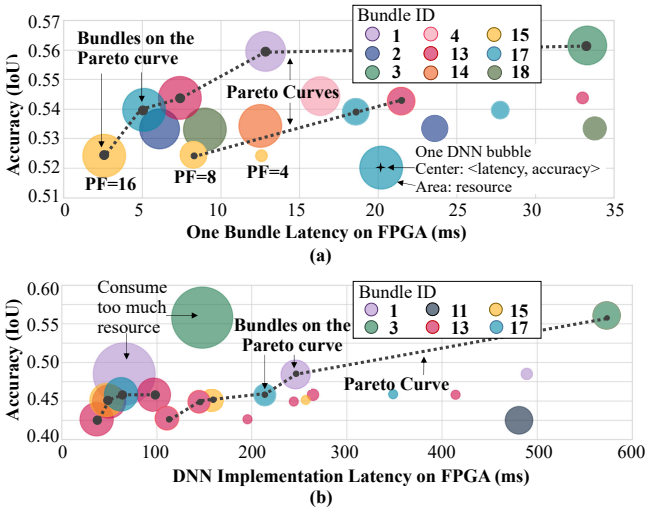


Figure 4: Coarse-grained bundle evaluation with (a) DNNs built using *method#1*; and (b) DNNs built using *method#2*.

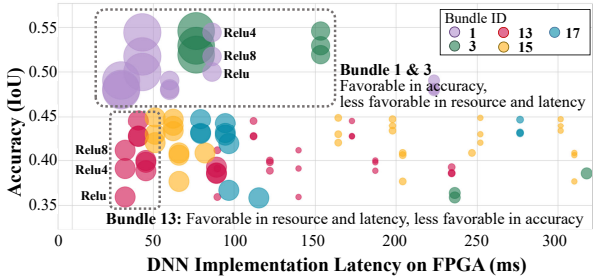


Figure 5: Fine-grained evaluation of the selected Bundles.

5.1 Bundle Evaluation and Selection

5.1.1 Coarse-Grained Evaluation. In this step, a three-dimensional feature including latency, resource and accuracy is captured for each Bundle. For latency and resource, we use Bundle and DNN modeling in Sec. 4.4; for accuracy, we train the DNNs built by Bundles on the target dataset. This evaluation is critical for co-design scalability, especially when a large number of Bundle candidates are provided for complex machine learning tasks.

We propose two methods to construct DNNs to evaluate Bundle accuracy. *method#1*: we use a DNN template with a fixed head and tail, and insert one Bundle replication in the middle; *method#2*: we replicate a Bundle for n times to build a DNN. Since Bundles may perform differently on various machine learning tasks, the constructed DNNs are directly trained on the target task in a *proxyless* manner [14]. For fast evaluation, each DNN is trained for a small number of epochs (20 in the experiment). After evaluation, Bundles with similar resource usage (e.g. DSPs) are grouped, and a Pareto curve is generated for each group. The Bundles on the Pareto curve will be selected.

Fig. 4 illustrates the coarse bundle evaluation on the example object detection task. Each bubble represents a DNN built from a Bundle. The coordinates of the bubble center represent latency and accuracy of the DNN, while the area of the bubble represents the resource usage. Under different parallel factors (PF), the implementation of a DNN differs in latency and resource but has the same accuracy. From Fig. 4 (a) and (b), we notice that both methods of

constructing DNNs can deliver similar results, where the Bundles on the Pareto curve are the same from both curves (Bundle 1, 3, 13, 15 and 17). It implies that our proposed Bundle evaluation is reliable for Bundle selection.

5.1.2 Fine-Grained Evaluation. After coarse-grained evaluation, a fine-grained evaluation on the selected Bundles is performed to better understand their characteristics. We construct DNNs by replicating certain Bundles for n times, and also try different activation functions such as *Relu*4 and *Relu*8, which relate to data quantization. Fig. 5 shows the fine-grained evaluation results for selected Bundles. It reveals that each Bundle has its own characteristics regarding latency, accuracy and resource overhead. For example, Bundle 1 and 3 are more promising in high accuracy DNNs with more resource and longer latency, while Bundle 13 is more favorable in DNNs targeting real-time responses with less resource.

5.2 Hardware-Aware DNN Search and Update

After selecting top-N promising Bundle candidates, *Auto-DNN* searches DNN models under resource and latency constraints. For each Bundle, K initial DNNs are generated and are incrementally updated until the latency target is met. Inside *Auto-DNN*, a Stochastic Coordinate Decent (SCD) unit is used for DNN update.

5.2.1 DNN Initialization. For each $bund_i$, total K DNNs will be generated, trained and fine-tuned as outputs. Output DNNs are denoted as DNN_i^k ($1 \le k \le K$), and each starts from an initial one denoted as $DNN_i^{k_0}$. First, we initialize software related variables. The $bund_i$ is replicated with N_i times; initial down sampling layers are inserted between replications; initial channel expansion factors are set to be 1 (do not expand) or 2 (double the number of channels), depending on the layer type. Next, hardware related variables will be traversed. Given $bund_i$, the IP templates, i.e., the IP_1 to IP_m in Table 1, are determined, and p_1 to p_m are instantiated. For simplicity, each IP template is instantiated into one p_j, configured with parallel factor PF_j and quantization scheme Q_j. We let Q_j and PF_j to be consistent among all IP instances to allow IP reuse across layers and BRAM buffer reuse across IPs. Under a certain Q_j, PF_j is set as the maximum value that can fully utilize available resources.

5.2.2 Stochastic Coordinate Descent (SCD) Unit. The SCD unit takes an initial $DNN_i^{k_0}$ as its input, together with a latency target L_{targ}, latency tolerance ϵ, and resource constraint Res_{max}. Denote the achieved latency of DNN_i^k as Lat and achieved resource as Res, the objective of SCD unit is $|Lat_{targ} - Lat| < \epsilon$ and $Res < Res_{max}$.

The SCD procedure is shown in Algorithm 1. Given an initial $DNN_i^{k_0}$, the SCD algorithm updates three variables: the number of Bundle replications, denoted as N_i; down-sampling configurations between bundles, denoted as X, which is a vector with zero-one entries indicating without/with down-samplings between Bundles; channel expansion configuration, denoted as Π, representing the vector $< f_{ch_1}, \cdots >$ in Table 1. The available channel expansion factors include $\{1.2, 1.3, 1.5, 1.75, 2\}$. Denote a unit *move* as Δ, the moves along three coordinates as Δ_N, Δ_Π and Δ_X, and the latency changes because of the moves as ΔLat_N, ΔLat_Π and ΔLat_X, respectively. Given the difference between Lat_{targ} and Lat as $\Delta L = |Lat_{targ} - Lat|$, the number of unit moves along N, Π and X directions are computed as $\Delta L/\Delta Lat_N$, $\Delta L/\Delta Lat_\Pi$ and $\Delta L/\Delta Lat_X$.

Algorithm 1 DNN Exploration with Stochastic Coordinate Decent

Input: Lat_{targ}, Lat tolerance ϵ, Res_{targ}, initial $DNN_i^{k_0}$
Output: K DNNs s.t. $|Lat_{targ} - Lat| < \epsilon$, $|Res < Res_{max}|$

Selected DNNs: $DNNs \leftarrow \emptyset$, initialize N, Π, $X \leftarrow DNN_i^{k_0}$
while $k < K$ **do**
  $Lat \leftarrow$ Est_Lat(DNN_i^k)
  if $|Lat_{targ} - Lat| < \epsilon$ **then**
    $k \leftarrow k + 1$, $DNNs \leftarrow DNNs \cup DNN_i^k$
  end if
  $\Delta Lat_N \leftarrow$ Est_Lat($(DNN[i_N + \Delta N])$)$-Lat$
  $\Delta Lat_\Pi \leftarrow$ Est_Lat($(DNN[i_\Pi + \Delta\Pi])$)$-Lat$
  $\Delta Lat_X \leftarrow$ Est_Lat($(DNN[i_X + \Delta X])$)$-Lat$
  Pick $\Delta \leftarrow \{\Delta N, \Delta\Pi, \Delta X\}$ uniformly at random
  if Est_Res($(DNN[i + \Delta])$) $< Res_{max}$ **then**
    if $\Delta = \Delta N$ **then** $\Delta_N \leftarrow \lfloor |Lat_{targ} - Lat|/\Delta Lat_N \rfloor$, $i_N \leftarrow i_N + \Delta_N$
    if $\Delta = \Delta\Pi$ **then** $\Delta_\Pi \leftarrow \lfloor |Lat_{targ} - Lat|/\Delta Lat_\Pi \rfloor$, $i_\Pi \leftarrow i_\Pi + \Delta_\Pi$
    if $\Delta = \Delta X$ **then** $\Delta_X \leftarrow \lfloor |Lat_{targ} - Lat|/\Delta Lat_X \rfloor$, $i_X \leftarrow i_X + \Delta_X$
  end if
  $DNN_i^k \leftarrow (DNN[i_N, i_\Pi, i_X])$
end while
return $DNNs$

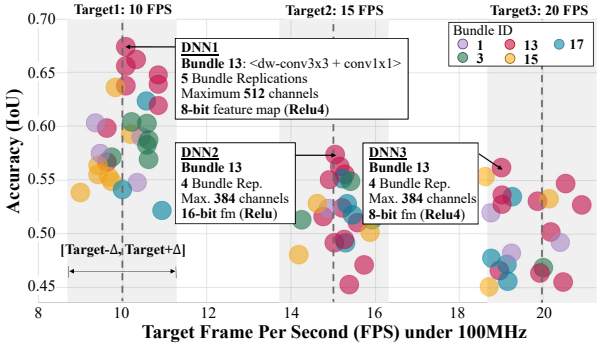


Figure 6: DNN models explored targeting 10/15/20 FPS @ 100MHz.

Then, the SCD algorithm picks one coordinate in random, and updates DNN_i^k along that direction within resource constraints.

When the objective of SCD is met, DNN_i^k is saved into set DNN_s as a candidate DNN. The K candidates are passed to DNN training framework to get their accuracy. Meanwhile, the DNNs are also passed to *Auto-HLS* to generate their FPGA implementations and get synthesized resource usage and latency.

5.2.3 Auto-HLS. To automatically generate FPGA accelerators for DNNs helps reduce the FPGA development cycle and engineering hours. Following the *Tile-Arch* template, *Auto-HLS* generates C code for FPGA accelerators, which can be directly synthesized by HLS tools. Since our IPs are written in C, knowing the input/output data dimensions of each IP and feature maps, the *Auto-HLS* generates function calls for the IPs with corresponding weight loading and data buffering functions. After C code generation, manual optimizations may be applied such as buffer re-allocation and loop fusion, which will be automated in the near future.

6 EXPERIMENTAL RESULTS

For demonstration, we use the same object detection task as in Sec. 5. To provide trade-off options between DNN latency and accuracy, we set three latency targets: 10, 15 and 20 FPS at 100MHz.

By specifying the resource constraints and latency targets, our proposed co-design methodology conducts DNN model exploration using selected Bundles, and outputs DNNs with their corresponding

Table 2: Performance Comparisons (FPGA and GPU competition data are obtained from [21])

	Model	IoU	Latency	FPS	Power	Energy	Efficiency	Resource Utilization			
								LUTs	DSP	BRAM	FF
Ours	DNN1	68.6%	80.0 ms (100 MHz)	12.5	2.2W	8.80 KJ	0.18 J/pic	82.5%	91.8%	96.1%	37.6%
			57.4 ms (150 MHz)	**17.4**	**2.5W**	**7.18 KJ**	**0.14 J/pic**	82.5%	91.8%	96.1%	37.6%
	DNN2	61.2%	62.6 ms (100 MHz)	16.0	2.2W	7.50 KJ	0.15 J/pic	76.4%	84.6%	77.9%	27.4%
			44.1 ms (150 MHz)	22.7	2.4W	5.51 KJ	0.11 J/pic	76.4%	84.6%	77.9%	27.4%
	DNN3	59.3%	47.8 ms (100 MHz)	20.9	2.2W	5.74 KJ	0.11 J/pic	70.4%	85.2%	95.4%	32.2%
			33.7 ms (150 MHz)	**29.7**	**2.4W**	**4.04 KJ**	**0.08 J/pic**	70.4%	85.2%	95.4%	32.2%
1st in FPGA	SSD	62.4%	84.6 ms (150 MHz)	11.96	4.2W	17.56 KJ	0.35 J/pic	83.9%	100%	78.9%	54.2%
2nd in FPGA	–	49.2%	38.5 ms (150 MHz)	25.97	2.5W	4.81 KJ	0.10 J/pic	88%	78%	77%	62%
3rd in FPGA	–	57.3%	136.1 ms (150 MHz)	7.35	2.6W	17.69 KJ	0.35 J/pic	63%	86%	95%	22%
1st in GPU	Yolo	69.8%	40.7 ms (854 MHz)	24.55	12.6W	25.66 KJ	0.51 J/pic	-	-	-	-
2nd in GPU	Tiny-Yolo	69.1%	39.5 ms (854 MHz)	25.3	13.3W	26.28 KJ	0.53 J/pic	-	-	-	-
3rd in GPU	Tiny-Yolo	68.5%	42.3 ms (854 MHz)	23.64	10.3W	21.79 KJ	0.44 J/pic	-	-	-	-

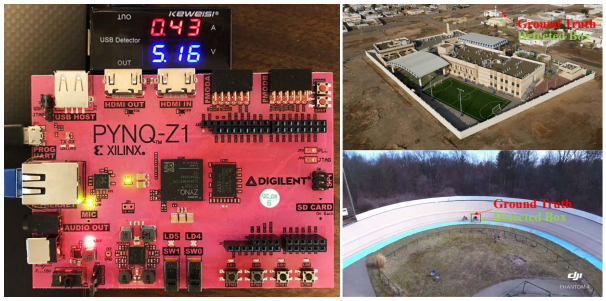


Figure 7: Pynq-Z1 board with power meter measured while running object detection.

accelerators. Fig. 6 shows all the explored DNNs that meet target latency within resource constraints. The DNNs which fall into the range $[target-\Delta, target+\Delta]$ are considered as candidates output for training. In total, 68 DNN models are built from 5 different Bundles with training and fine-tuning. Among them, we pick those with the best accuracy for each FPS target and get DNN1~3. The detailed structures of the final DNNs are shown in Fig. 6. DNN1 achieves the highest IoU, reaching 68.6% with 12.5 FPS@100MHz and 17.4 FPS@150MHz. DNN2 achieves 61.2% IoU with 16.0 FPS@100MHz and 22.7 FPS@150MHz, while DNN3 achieves the highest FPS at 29.7 FPS@150MHz with 59.3% IoU. Some additional modifications are applied on the *Auto-HLS* generated C code, such as on-chip buffer allocation and loop fusion, to reach higher FPS.

We also compare to the state-of-the-art works for this object detection task on PYNQ-Z1 published in [21]. The comparisons to FPGA and GPU categories are shown in Table 2. The results are collected from the board-level implementations. The IoU is measured on 50K images from the official dataset following the same criteria in DAC-SDC. Latency refers to a single frame latency, while FPS is measured using total run-time for the 50K images including image loading, preprocessing, and DNN inference. The power and energy are measured using the POWER-Z KT001 USB Power Monitor as shown in Fig. 7. We also show two example images with the ground truth bounding boxes (red) and our generated boxes (green).

Compared to the 1st-place winner of the FPGA category, we achieve 6.2% higher IoU, 40% lower power, and 2.5× better energy efficiency. The 1st-place FPGA team follows the top-down design flow by starting from a standard DNN-based detector (SSD). After network compression, the DNN is small enough that satisfies both hardware constraints and performance demands [22]. Compared to this top-down approach, our co-design method is able to deliver better DNN models and more efficient hardware accelerators. Compared to GPU-based designs, our DNN1 model is more accurate than the 3rd-place design and only 1.2% lower IoU than the 1st-place GPU design. Regarding the energy efficiency, ours is 3.6× better than the 1st-place GPU design with 40% longer latency despite a nearly 6× slower clock frequency.

7 CONCLUSION

We presented an FPGA/DNN co-design methodology with both bottom-up DNN model exploration and top-down accelerator design approaches to enhance the IoT intelligence on embedded FPGAs. On the defined co-design space, we proposed *Auto-DNN*, an automatic DNN model search engine to explore hardware-friendly DNNs, and an automatic HLS generator, *Auto-HLS*, to generate FPGA-based DNN accelerators. We applied our proposed methodology to an object detection task from DAC-SDC competition. Results showed that our implementation outperformed the 1st place winner in all factors with 6.2% higher IoU, 40% lower power, and 2.5× better energy efficiency. Comparing to GPU designs, our results achieved similar accuracy (0.1% better than 3rd place and 1.2% worse than 1st place) but with 3.1× to 3.8× better energy efficiency.

ACKNOWLEDGMENTS

This work was partly supported by the IBM-Illinois Center for Cognitive Computing System Research (C^3SR) – a research collaboration as part of IBM AI Horizons Network.

REFERENCES

[1] Chen Zhang et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.
[2] Jiantao Qiu et al. Going deeper with embedded FPGA platform for convolutional neural network. In *FPGA*, 2016.
[3] Xiaofan Zhang et al. High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *FPL*, 2017.
[4] Junsong Wang et al. Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA. In *FPL*, 2018.
[5] Xiaofan Zhang et al. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *ICCAD*, 2018.
[6] Qin Li et al. Implementing neural machine translation with bi-directional gru and attention mechanism on FPGAs using HLS. In *ASP-DAC*, 2019.
[7] Xiaofan Zhang et al. Machine learning on FPGAs to face the IoT revolution. In *ICCAD*, 2017.
[8] Barret Zoph et al. Learning transferable architectures for scalable image recognition. *arXiv:1707.07012*, 2017.
[9] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578*, 2016.
[10] Deming Chen et al. Lopass: A low-power architectural synthesis system for fpgas with interconnect estimation and optimization. *IEEE TVLSI*, 18(4):564–577, 2010.
[11] Kyle Rupnow et al. High level synthesis of stereo matching: Productivity, performance, and software constraints. In *FPT*, 2011.
[12] Esteban Real et al. Regularized evolution for image classifier architecture search. *arXiv:1802.01548*, 2018.
[13] Mingxing Tan et al. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv:1807.11626*, 2018.
[14] Han Cai et al. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv:1812.00332*, 2018.
[15] Song Han et al. Ese: Efficient speech recognition engine with sparse LSTM on FPGA. In *FPGA*, 2017.
[16] Mohammad Motamedi et al. Design space exploration of FPGA-based deep convolutional neural networks. In *ASP-DAC*, 2016.
[17] Guanwen Zhong et al. Design space exploration of FPGA-based accelerators with multi-level parallelism. In *DATE*, 2017.
[18] Kaiming He et al. Deep residual learning for image recognition. In *CVPR*, 2016.
[19] Mark Sandler et al. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.
[20] 2018 DAC System Design Contest. http://www.cse.cuhk.edu.hk/~byu/2018-DAC-HDC/ranking.html#may. Accessed: 2018-09-04.
[21] Xiaowei Xu et al. DAC-SDC low power object detection challenge for UAV applications. *arXiv:1809.00110*, 2018.
[22] 1st Place in FPGA Category. https://github.com/hirayaku/DAC2018-TGIIF. Accessed: 2018-10-01.